CS251 Fall 2022

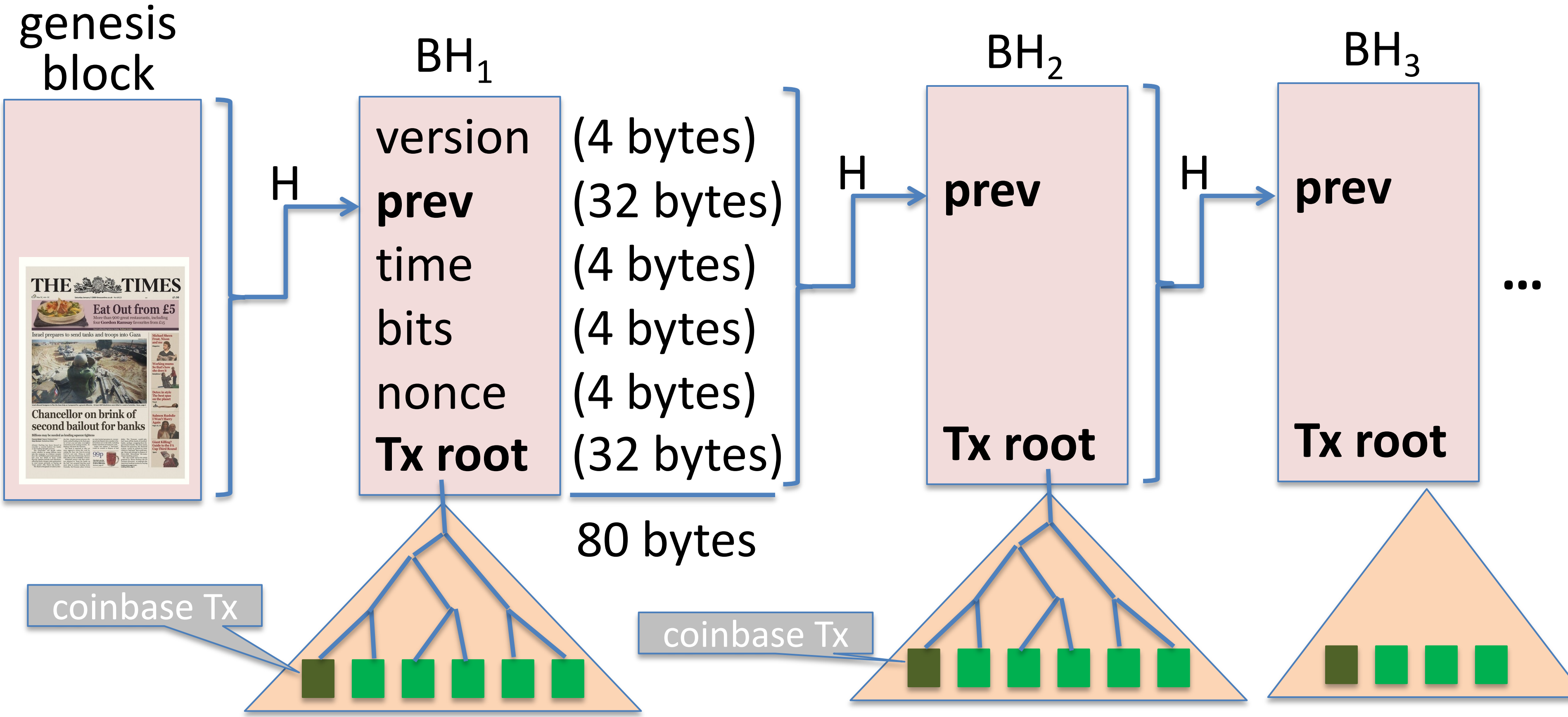
(cs251.stanford.edu)

Bitcoin Scripts and Wallets

Dan Boneh

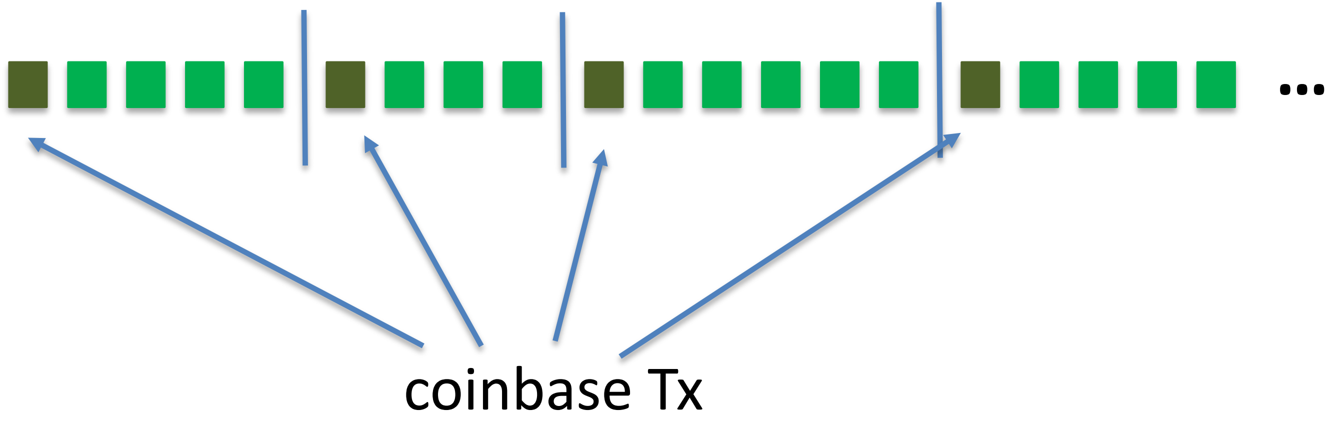
Note: HW#1 is posted on the course web site. Due Tue, Oct. 11.

Recap: the Bitcoin blockchain



Tx sequence

View the blockchain as a sequence of Tx (append-only)



Tx cannot be erased: mistaken Tx $\Rightarrow$ locked or lost of funds

Tx structure (non-coinbase)

inputs	input[0] input[1] input[2]
outputs	output[0] output[1]
(segwit)	witnesses (part of input)
(4 bytes)	locktime

TxID = H(Tx)
(excluding witnesses)

earliest block # that can include Tx

input:

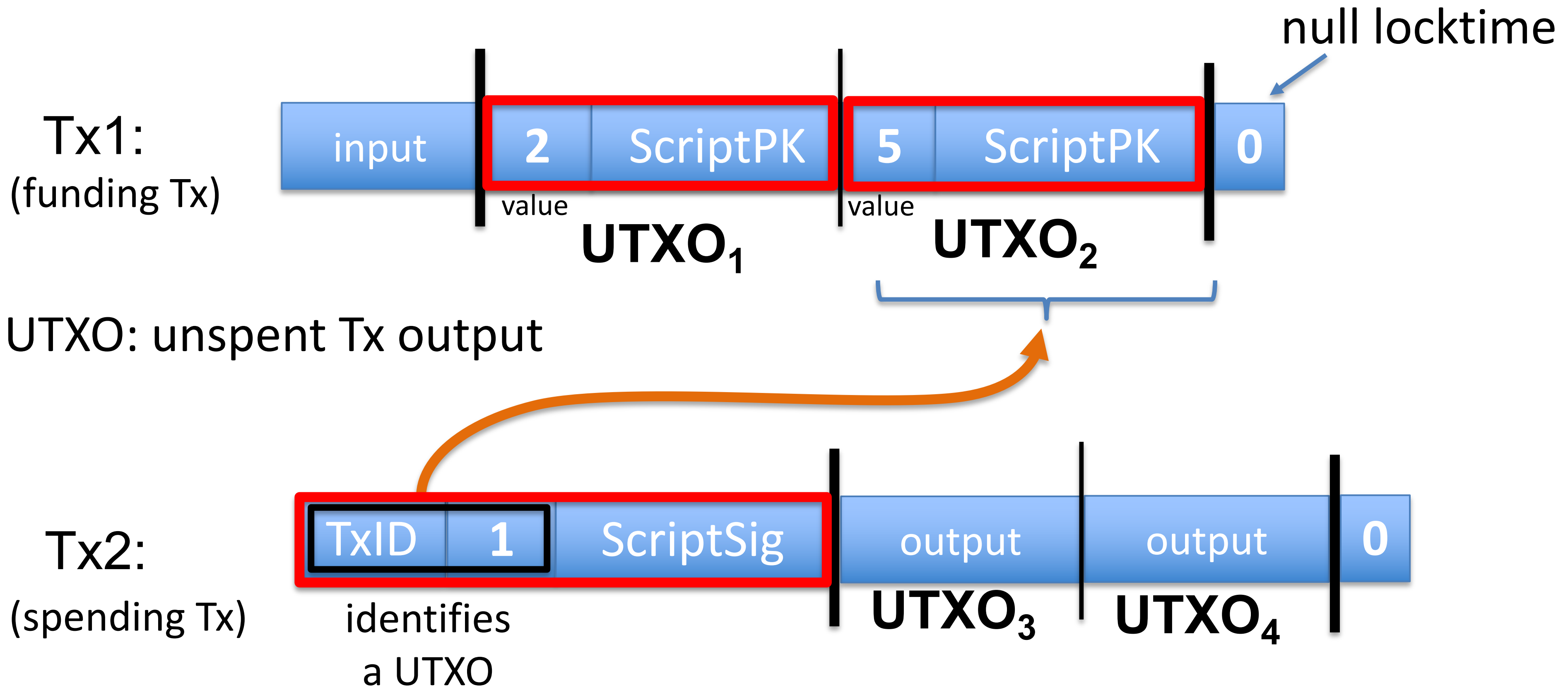
TxID	32 byte hash
out-index	4 byte index
ScriptSig	program
seq	ignore

output:

value	8 bytes
ScriptPK	program

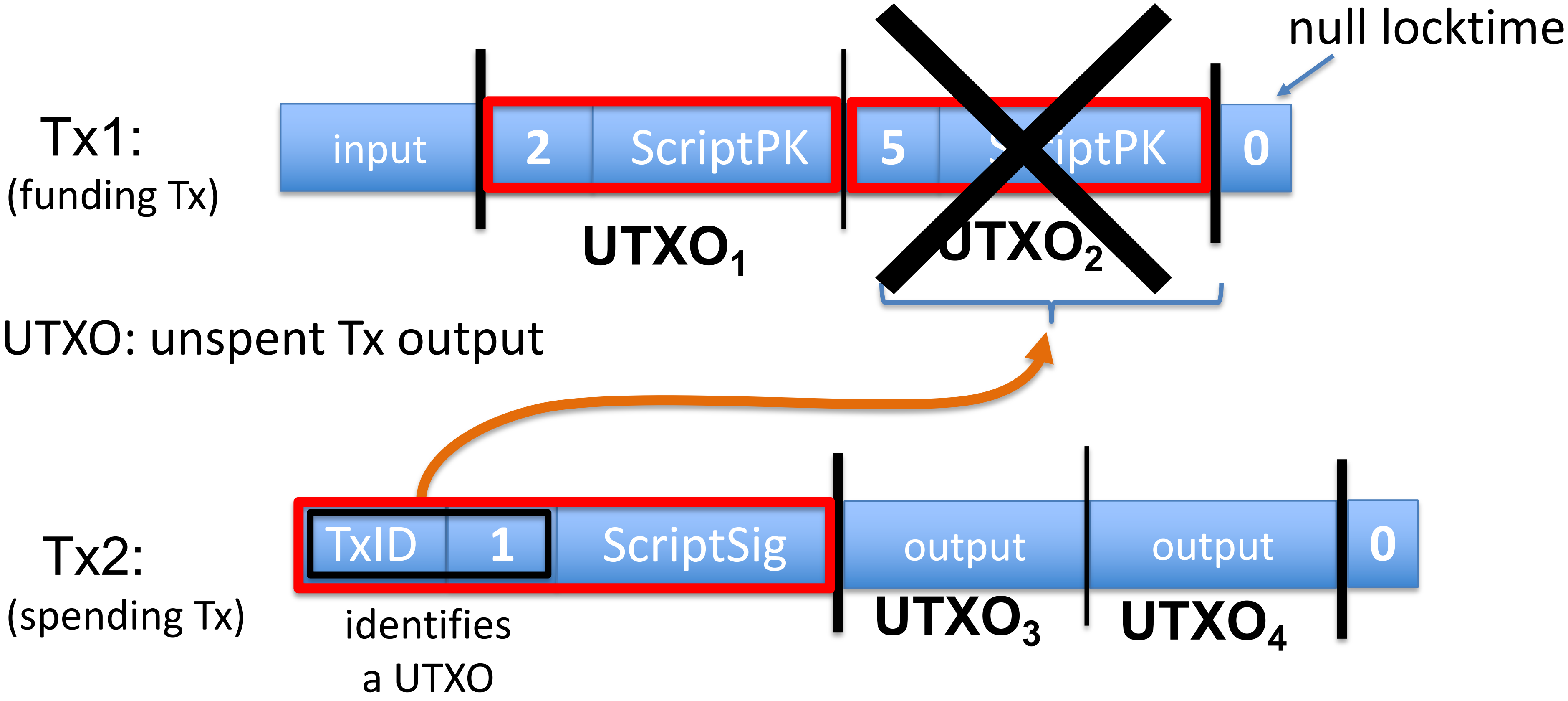
$\#BTC = value/10^8$

Example



null locktime
Tx1:
(funding Tx)
input
2
ScriptPK
5
ScriptPK
0
value
value
$\mathbf{UTXO_1}$
$\mathbf{UTXO_2}$
UTXO: unspent Tx output
Tx2:
(spending Tx)
TxID
1
ScriptSig
output
output
0
identifies
a UTXO
$\mathbf{UTXO_3}$
$\mathbf{UTXO_4}$

Example


null locktime
Tx1:
(funding Tx)
input
2
ScriptPK
5
ScriptPK
0
$UTXO_1$
$UTXO_2$
UTXO: unspent Tx output
Tx2:
(spending Tx)
TxID
1
ScriptSig
output
output
0
$UTXO_3$
$UTXO_4$
identifies
a UTXO

Validating Tx2

Miners check (for each input):

program from funding Tx:
under what conditions
can UTXO be spent

1. The program ScriptSig | ScriptPK returns true

program from spending Tx:
proof that conditions
are met

2. TxID | index is in the current UTXO set

3. sum input values ≥ sum output values

After Tx2 is posted, miners remove $UTXO_2$ from UTXO set

Transaction types: (1) P2PKH

pay to public key hash

Alice want to pay Bob 5 BTC:

- step 1: Bob generates sig key pair $(pk_B, sk_B) \leftarrow Gen()$
- step 2: Bob computes his Bitcoin address as $addr_B \leftarrow H(pk_B)$
- step 3: Bob sends $addr_B$ to Alice
- step 4: Alice posts Tx:

Point to Alice's UTXO

	UTXO$_B$ for Bob		UTXO$_A$ for Alice (change)		
input 7 BTC	**5**	ScriptPK$_B$	**2**	ScriptPK$_A$	**0**

ScriptPK$_B$: DUP HASH256 **<$addr_B$>** EQVERIFY CHECKSIG

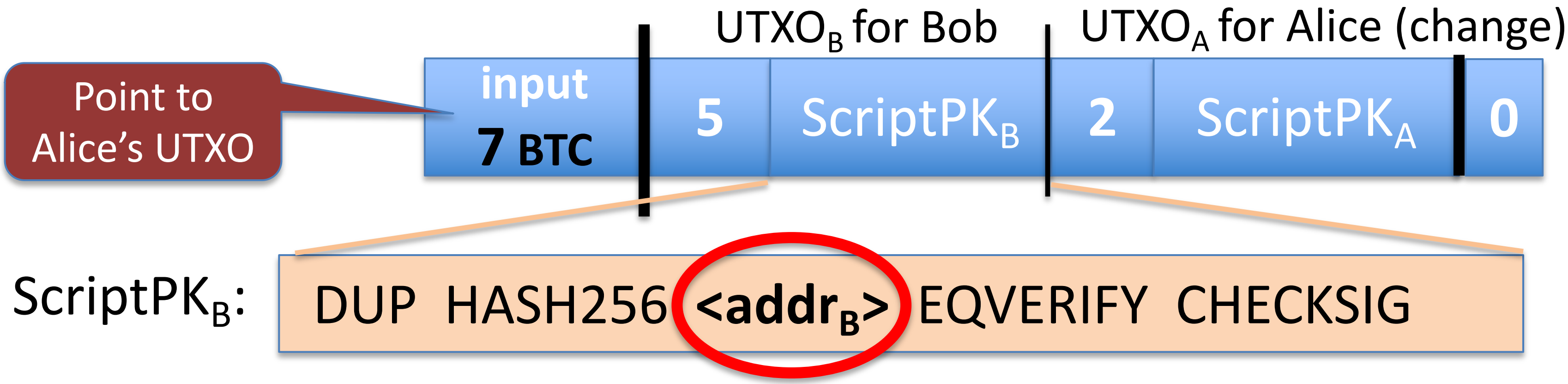
Transaction types: (1) P2PKH

pay to public key hash

"input" contains ScriptSig that authorizes spending Alice's UTXO

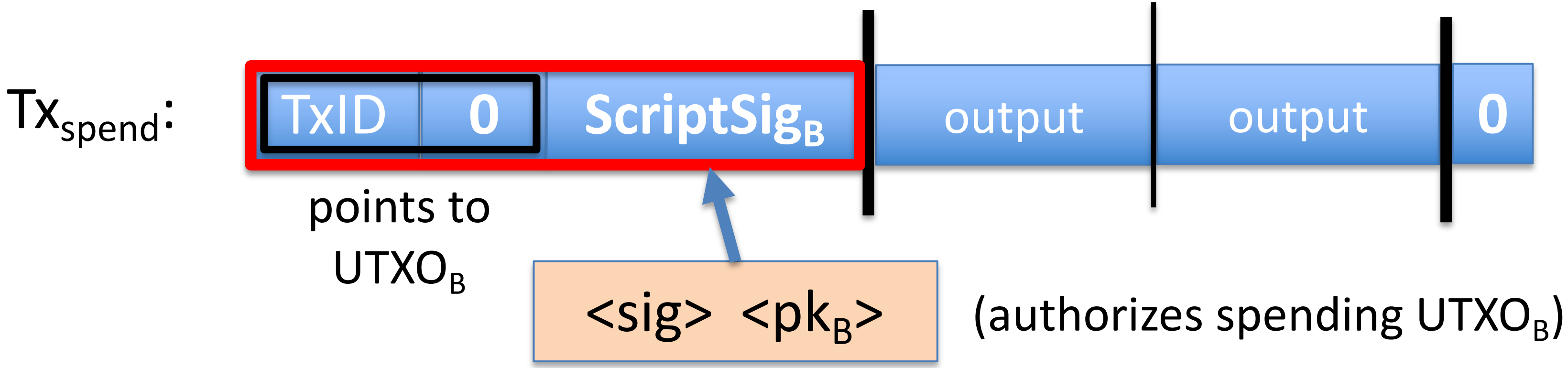
- example: ScriptSig contains Alice's signature on Tx

  $\Longrightarrow$ miners cannot change ScriptPK$_B$ (will invalidate Alice's signature)



Transaction types: (1) P2PKH

Later, when Bob wants to spend his UTXO: create a Tx_{spend}



<sig> = Sign(sk_B, Tx) where Tx = (Tx_{spend} excluding all ScriptSigs) (SIGHASH_ALL)

Miners validate that $\boxed{ScriptSig_B \mid ScriptPK_B}$ returns true

P2PKH: comments

- Alice specifies recipient's pk in $UTXO_B$

- Recipient's pk is not revealed until UTXO is spent

  (some security against attacks on pk)

- Miner cannot change <$Addr_B$> and steal funds:

  invalidates Alice's signature that created $UTXO_B$

Segregated Witness

ECDSA malleability:

Given (m, sig) anyone can create (m, sig') with sig $\neq$ sig'

$\Rightarrow$ miner can change sig in Tx and change TxID = SHA256(Tx)

$\Rightarrow$ Tx issuer cannot tell what TxID is, until Tx is posted

$\Rightarrow$ leads to problems and attacks

Segregated witness: signature is moved to witness field in Tx

TxID = Hash(Tx without witnesses)

Transaction types: (2) P2SH: pay to script hash

(pre SegWit in 2017)

Payer specifies a redeem script (instead of just pkhash)

Usage: (1) Bob publishes hash(redeem script) ⟵ Bitcoint addr.

(2) Alice sends funds to that address in funding Tx

(3) Bob can spend UTXO if he can satisfy the script

ScriptPK in UTXO: HASH160 <H(redeem script)> EQUAL

ScriptSig to spend: $\langle sig_1 \rangle$ $\langle sig_2 \rangle$ … $\langle sig_n \rangle$ <redeem script>

payer can specify complex conditions for when UTXO can be spent

P2SH

Miner verifies:

(1) <ScriptSig> ScriptPK = true ⟵ spending Tx gave correct script

(2) ScriptSig = true ⟵ script is satisfied

Example P2SH: multisig

<u>Goal</u>: spending a UTXO requires t-out-of-n signatures

Redeem script for 2-out-of-3: (chosen by payer)

$\langle 2\rangle \ \langle PK_1\rangle \ \langle PK_2\rangle \ \langle PK_3\rangle \ \langle 3\rangle$ CHECKMULTISIG

threshold

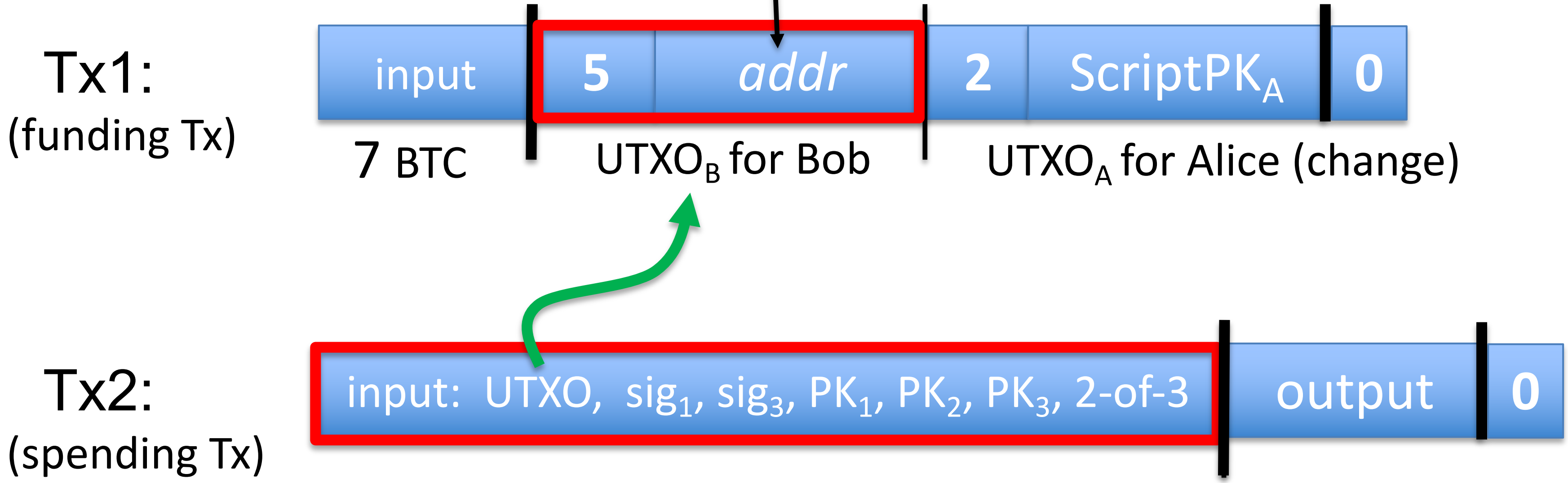
hash gives P2SH address

ScriptSig to spend: (by payee) <0> <sig1> <sig3> <redeem script>

(in the clear)

Abstractly …

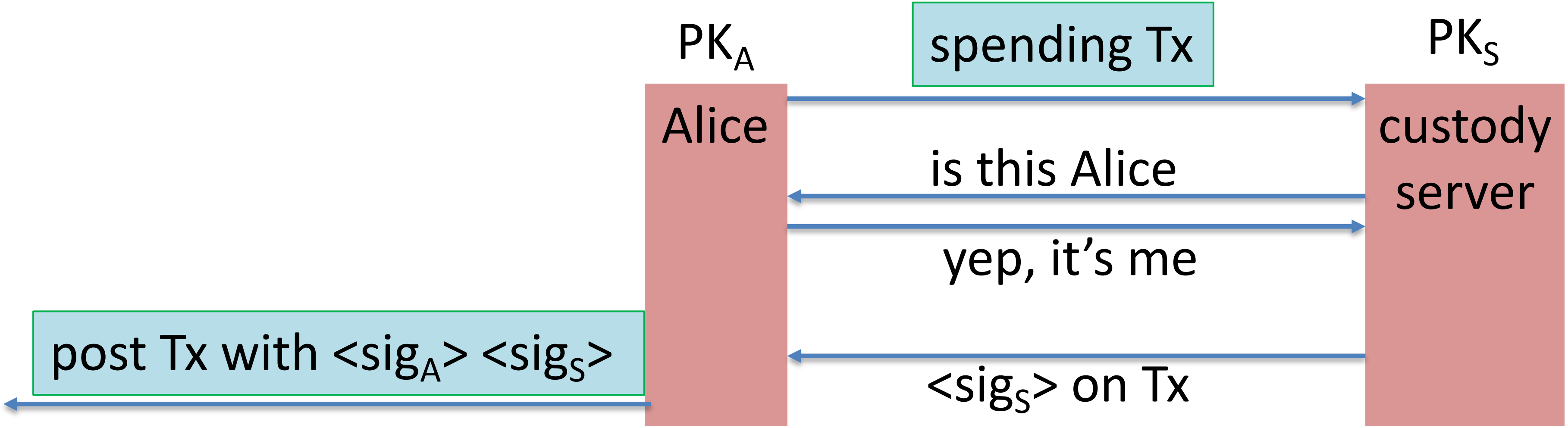
Multisig address: $addr = H(PK_1, PK_2, PK_3, \text{2-of-3})$



Example Bitcoin scripts

Protecting assets with a co-signatory

Alice stores her funds in UTXOs for $\boxed{\textit{addr} = \textbf{2-of-2}(\textbf{PK}_A, \textbf{PK}_S)}$



$\Rightarrow$ theft of Alice's SK_A does not compromise BTC

Escrow service

Alice wants to buy a backpack for 0.1฿ from merchant Bob

Goal: Alice only pays after backpack arrives, but can't not pay

addr = **2-of-3(PK_A, PK_B, PK_J)**

post payment of 0.11฿ to *addr*

(creates $UTXO_A$)

Alice

PK_A

want backpack for 0.1฿

once see Tx on chain

mail backpack

backpack arrives

send <sig_A> on Tx:

$UTXO_A \rightarrow (PK_B{:}0.1,\ PK_A{:}0.01)$

Bob

PK_B

redeem using <sig_A> <sig_B> on Tx

Judge

PK_J

Escrow service: a dispute

(1) Backpack never arrives: (Bob at fault)

Alice gets her funds back with help of Judge and a Tx:

Tx: ($\mathbf{UTXO_A \rightarrow PK_A}$, $\mathbf{sig_A}$, $\mathbf{sig_{Judge}}$) [2-out-of-3]

(2) Alice never sends sig_A: (Alice at fault)

Bob gets paid with help of Judge as a Tx:

Tx: ($\mathbf{UTXO_A \rightarrow PK_B}$, $\mathbf{sig_B}$, $\mathbf{sig_{Judge}}$) [2-out-of-3]

(3) Both are at fault: Judge publishes <sig_{Judge}> on Tx:

Tx: ($\mathbf{UTXO_A \rightarrow PK_A: 0.05,\ PK_B: 0.05,\ PK_J: 0.01}$)

Now either Alice or Bob can execute this Tx.

Cross Chain Atomic Swap

Alice has 5 BTC, Bob has 2 LTC (LiteCoin). They want to swap.

Want a sequence of Tx on the Bitcoin and Litecoin chains s.t.:

- either success: Alice has 2 LTC and Bob has 5 BTX,
- or failure: no funds move.

Swap cannot get stuck halfway.

Goal: design a sequence of Tx to do this.

solution: programming proj #1 ex 4.

Managing crypto assets: Wallets

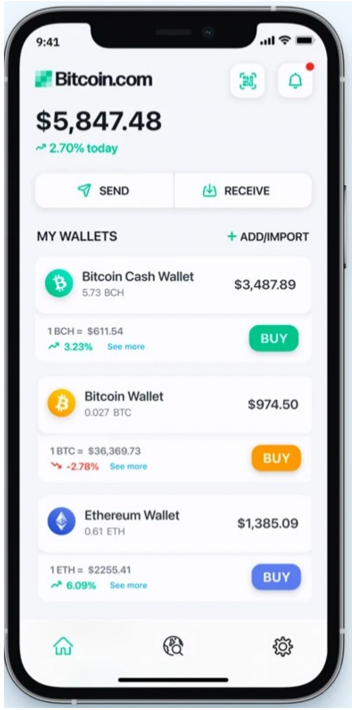
Managing secret keys

Users can have many PK/SK:

- one per Bitcoin address, Ethereum address, …

Wallets:

- Generates PK/SK, and stores SK,
- Post and verify Tx,
- Show balances



Managing lots of secret keys

Types of wallets:

- **cloud** (e.g., Coinbase): cloud holds secret keys … like a bank.
- **laptop/phone**: Electrum, MetaMask, …
- **hardware**: Trezor, Ledger, Keystone, …
- **paper**: print all sk on paper
- **brain**: memorize sk (bad idea)

client stores secret keys

- **Hybrid**: non-custodial cloud wallet (using threshold signatures)

Not your keys, not your coins … but lose key ⇒ lose funds

Simplified Payment Verification (SPV)

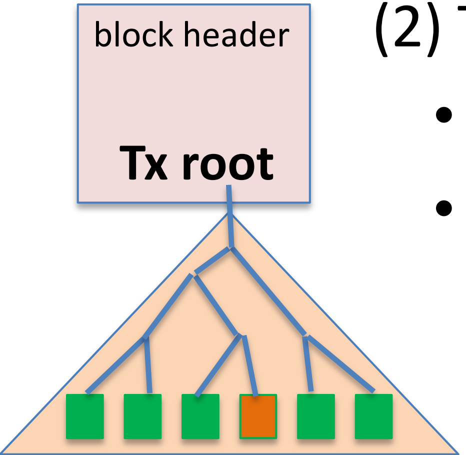
How does a client wallet display Alice’s current balances?

- Laptop/phone wallet needs to verify an incoming payment
- **Goal**: do so w/o downloading entire blockchain (366 GB)

SPV: (1) download all block headers (60 MB)

(2) Tx download:

- wallet → server: list of my wallet addrs (Bloom filter)
- server → wallet: Tx involving addresses + Merkle proof to block header.



Simplified Payment Verification (SPV)

Problems:

(1) **Security**: are BH the ones on the blockchain? Can server omit Tx?

- Electrum: download block headers from ten random servers, optionally, also from a trusted full node.

List of servers: electrum.org/#community

(2) **Privacy**: remote server can test if an *addr* belongs to wallet

We will see better light client designs later in the course (e.g. Celo)

Hardware wallet: Ledger, Trezor, …

End user can have lots of secret keys. How to store them ???

Hardware wallet (e.g., Ledger Nano X)

- connects to laptop or phone wallet using Bluetooth or USB
- manages many secret keys
  - Bolos OS: each coin type is an app on top of OS
- PIN to unlock HW (up to 48 digits)
- screen and buttons to verify and confirm Tx

Hardware wallet: backup

Lose hardware wallet $\Rightarrow$ loss of funds. What to do?

Idea 1: generate a secret seed $k_0 \in \{0,1\}^{256}$

for i=1,2,…: $sk_i \leftarrow HMAC(k_0, i)$, $pk_i \leftarrow g^{sk_i}$

ECDSA public key

$pk_1, pk_2, pk_3, \ldots$: random unlinkable addresses (without k_0)

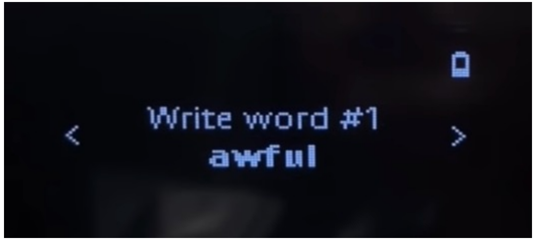
k_0 is stored on HW device and in offline storage (as 24 words)

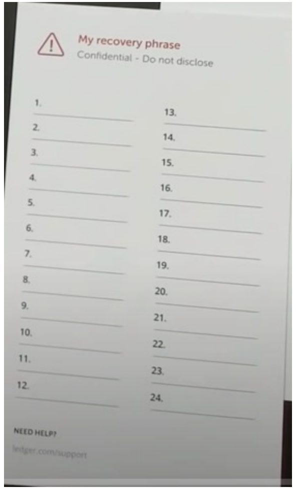
$\Rightarrow$ in case of loss, buy new device, restore k_0, recompute keys

On Ledger

When initializing ledger:

- user asked to write down the 24 words
- each word encodes 11 bits (24 × 11 = 268 bits)
  - list of 2048 words in different languages (BIP 39)

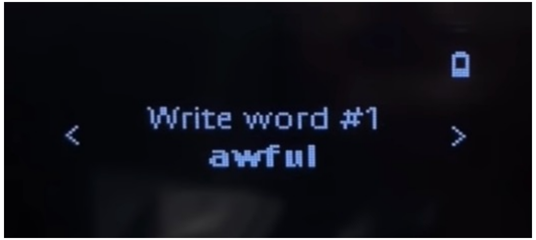




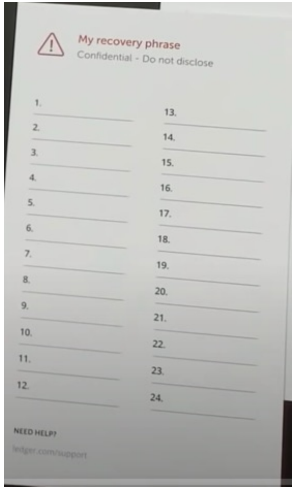
Example: English word list

2048 lines (2048 sloc) | 12.8 KB

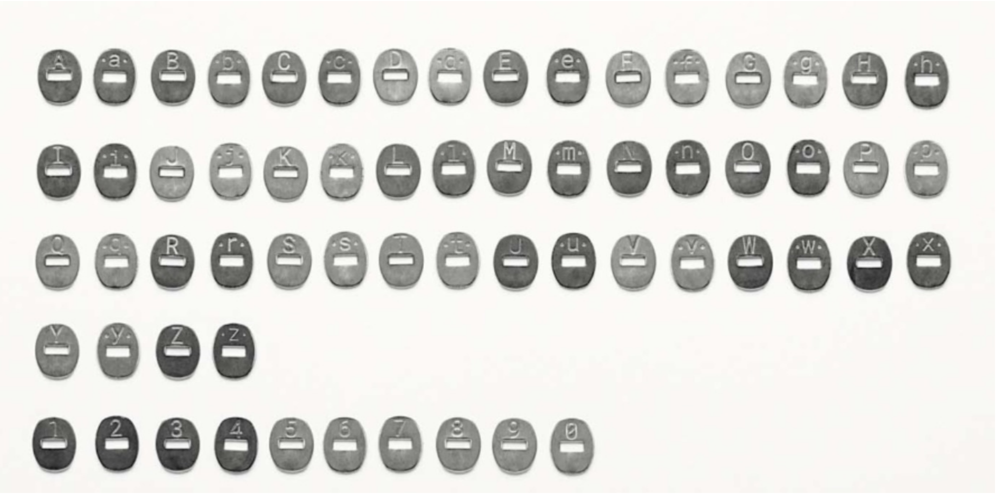
1	abandon
2	ability
3	able
4	about
5	above
6	absent
7	absorb
8	abstract
9	absurd
10	abuse
	⋮
2046	zero
2047	zone
2048	zoo



save list of
24 words



Crypto Steel

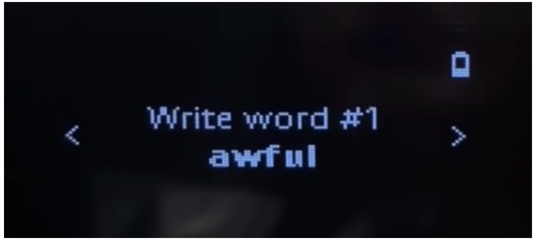


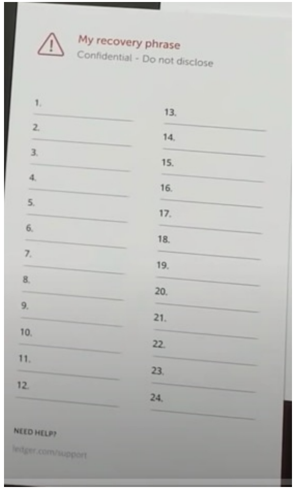
Careful with unused letters …

On Ledger

When initializing ledger:

- user asked to write down the 24 words
- each word encodes 11 bits (24 × 11 = 268 bits)
    - list of 2048 words in different languages (BIP 39)





Beware of "pre-initialized HW wallet"

- 2018: funds transferred to wallet promptly stolen

How to securely check balances?

With Idea1: need k_0 just to check my balance:

- k_0 needed to generate my addresses $(pk_1, pk_2, pk_3, \ldots)$

  … but k_0 can also be used to spend funds

- Can we check balances without the spending key ??

Goal: two seeds

- k_0 lives on Ledger: can generate all secret keys (and addresses)
- k_{pub}: lives on laptop/phone wallet: can only generate addresses (for checking balance)

Idea 2: (used in HD wallets)

secret seed: $k_0 \in \{0,1\}^{256}$; $(k_1, k_2) \leftarrow$ HMAC(k_0, "init")

balance seed: $k_{pub} = (k_2,\ \ h = g^{k_1})$

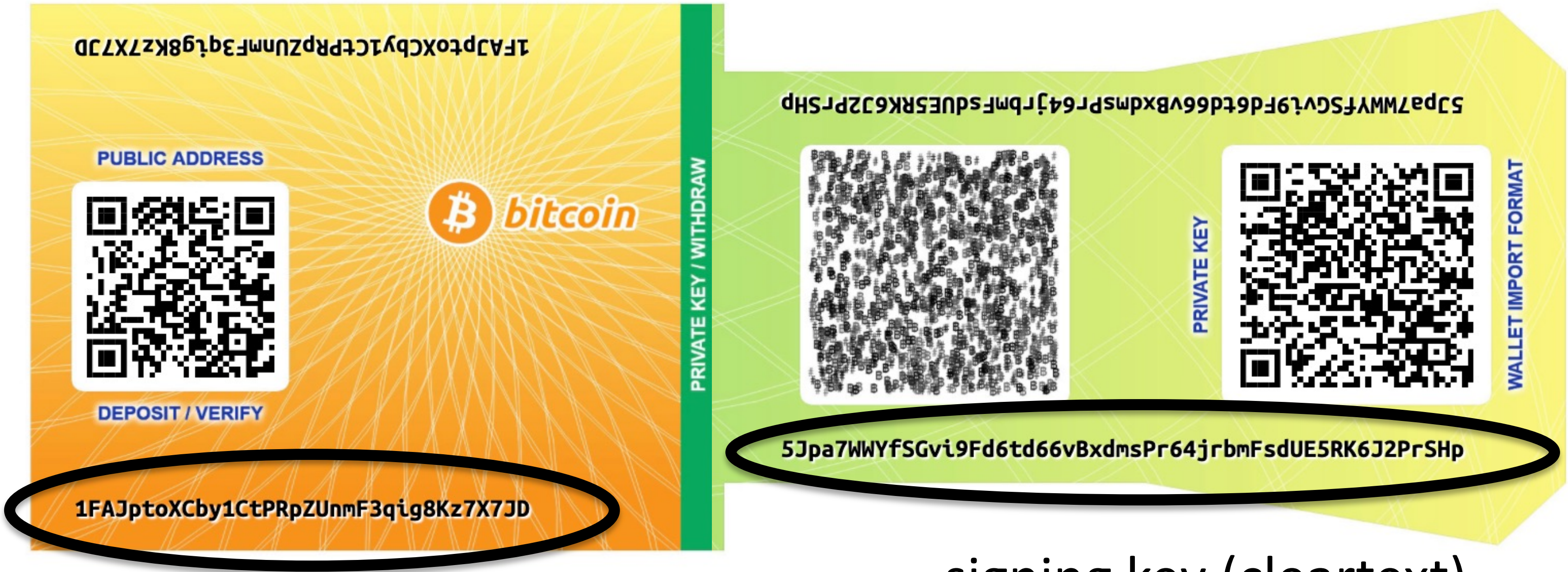
for all i=1,2,...: $\begin{cases} sk_i \leftarrow k_1 + \text{HMAC}(k_2, i) \\ pk_i \leftarrow g^{sk_i} = g^{k_1} \cdot g^{HMAC(k_2,i)} = \underbrace{h \cdot g^{HMAC(k_2,i)}}_{\text{computed from } k_{pub}} \end{cases}$

k_{pub} does not reveal sk_1, sk_2, ...

k_{pub}: on laptop/phone, generates unlinkable addresses pk_1, pk_2, ...

k_0: on ledger

Paper wallet (be careful when generating)



Bitcoin address = base58(hash(PK))

signing key (cleartext)

base58 = a-zA-Z0-9 without {0,O,l,1}

Managing crypto assets in the cloud

How exchanges store assets

Hot/cold storage

Coinbase: holds customer assets

Design: 98% of assets (SK) are held in cold storage

cold storage (98%)

$\boxed{k_0^{(1)}}$

$\boxed{k_0^{(2)}}$

$\boxed{k_0^{(3)}}$

$\boxed{k_0}$

t-out-of-n secret sharing of k_0

hot wallet (2%)

h, k_2	$\mathrm{SK_{hot}}$
used to verify cold storage balances	2% of assets

customers

Problems

Can’t prove ownership of assets in cold storage,
without accessing cold storage:

- To prove ownership (e.g., in audit or in a proof of solvency)
- To participate in proof-of-stake consensus

Solutions:

- Keep everything in hot wallet (e.g, Anchorage)
- Proxy keys: keys that prove ownership of assets,
  but cannot spend assets

END OF LECTURE

Next lecture: consensus